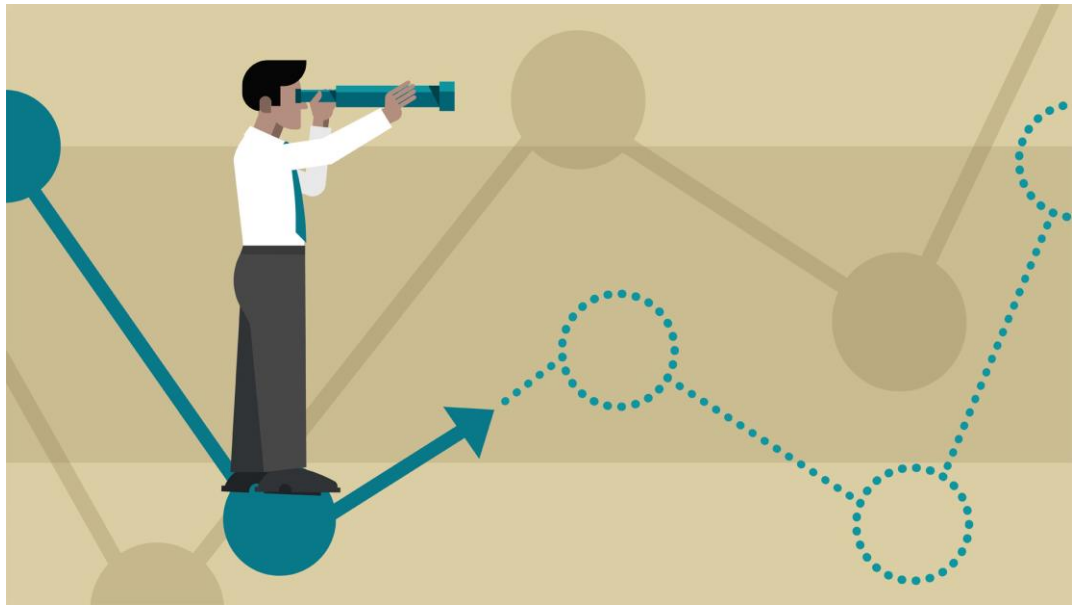
Forecasting through Artificial Neural Networks

Business Intelligence per i Servizi Finanziari 2023-2024

Antonio Candelieri

Forecasting

- In the analysis of financial time series we are interested in designing models for **predicting future values**.



Artificial Neural Networks… in brief

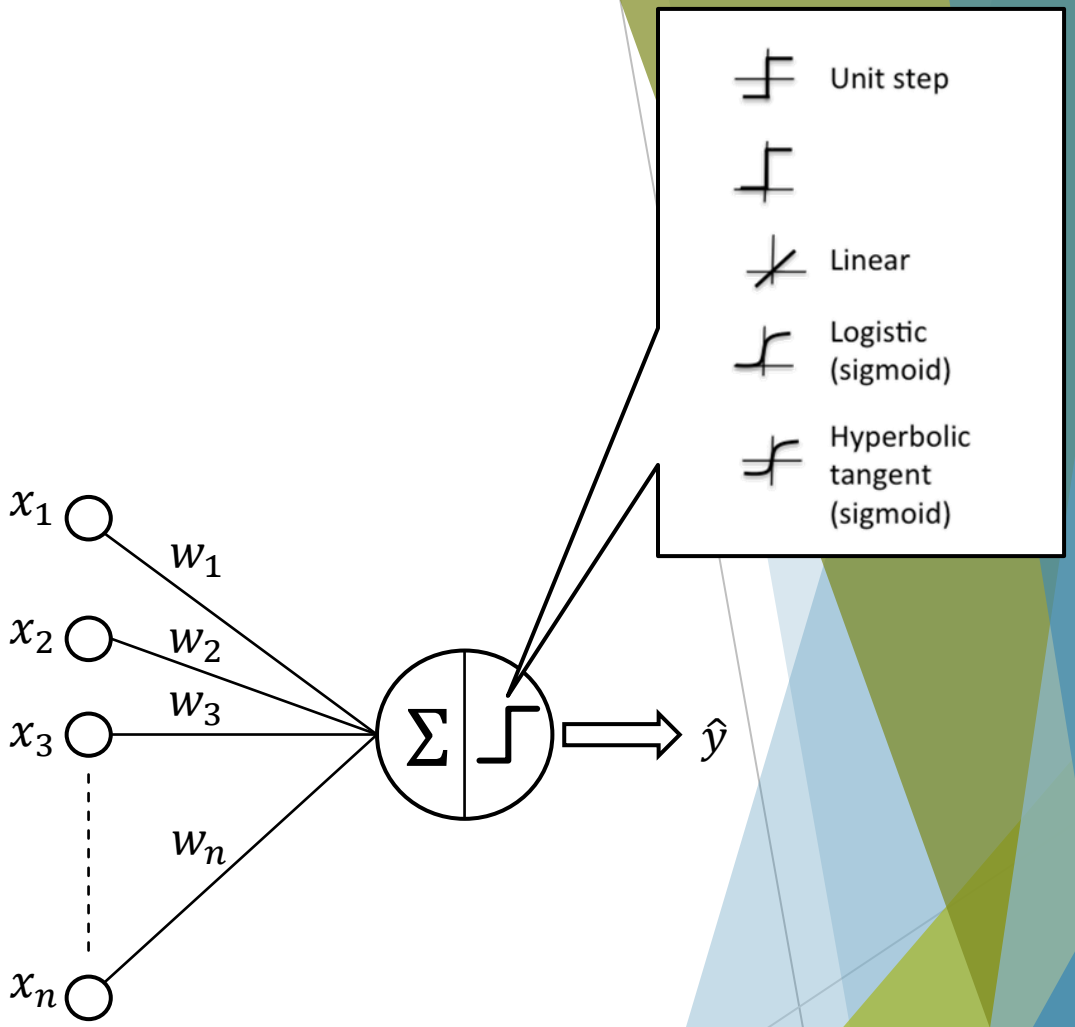
- Simulating the human brain, which consists of cells (***neurons***) interconnected through ***synapsis***
- ***Connectionist learning*** paradigm: changing the strenght of synaptic connections to reduce error on a specific task
- Neurons are very simple (computational) units: the power of a brain is given by the organization of the connections among units (i.e., ***architecture***)

The Perceptron

- n Input neurons x_i with $i = 1, .., n$
- 1 computational neuron (Perceptron)

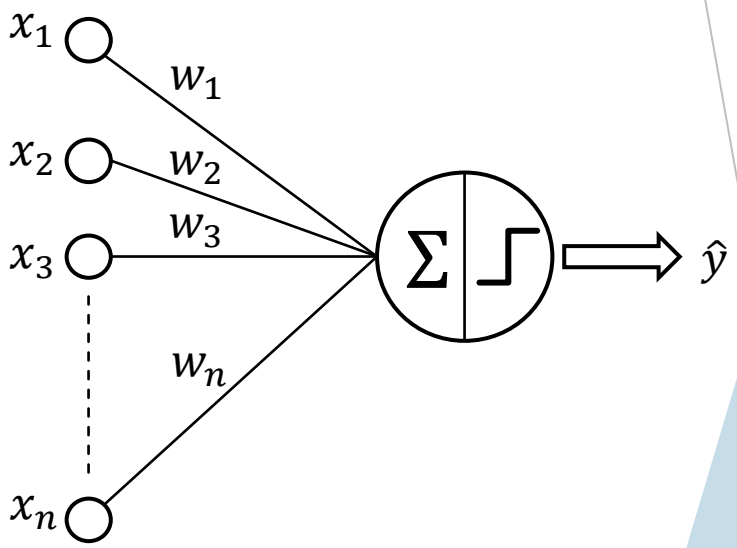
- n weights w_i with $i = 1, .., n$

- Perceptron:
  - Weighted sum of inputs $(\boldsymbol{\Sigma})$
  - ***Activation function***
  - 1 outcome



The Perceptron: learning algo

- Searching for w_i, with $i = 1,..,n$, such that the error between **predicted** and **actual** output ($\hat{y}$ and y) is minimized
- Therefore, we must know x_i, with $i = 1,..,n$, and the associated y (i.e., Supervised Learning setting)
- In forecasting we have $y = X_{t+h}$ and $x_i = X_{t-i+1}$
- Surely, more than just an example (*instance*) is needed to learn an accurate predicitive model. A dataset of N must be available: $D = \{(\mathbf{x}^{(j)}, y^{(j)})\}_{j=1,\dots,N}, \mathbf{x}^{(j)} \in \mathbb{R}^n$



Just a remind from linearity assumption

- A linear predictor is defined as

$$L(X_{t+h}) = \sum_{j=0}^{p} \alpha_j X_{t-j}$$

- With coefficients (i.e., parameters) α_j tuned in order to minimize the prediction error

$$F(\alpha_0, \ldots, \alpha_p) = E\left(X_{t+h} - \sum_{j=0}^{p} \alpha_j X_{t-j}\right)^2$$

- Under linearity assumption, *F* is a quadratic function bounded below by 0, and hence there exists values of $(a_0, \ldots, a_p)$ that minimizes *F*, and this minimum satisfies

$$\frac{\partial F(\alpha_0, \ldots, \alpha_p)}{\partial \alpha_j} = 0, j = 0, \ldots, p.$$

Perceptron is a non-linear predictor

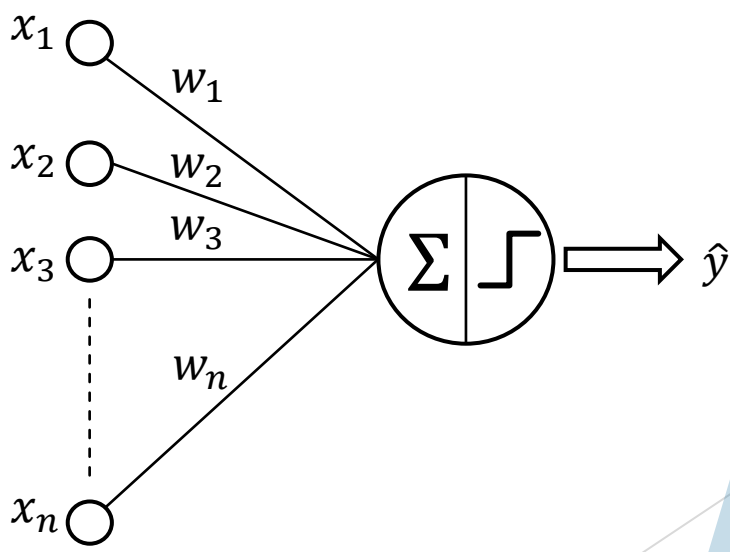
- The concept is similar in training a Perceptron but, depedning on the activation function, the predictor is non-linear:

$$\hat{y} = \mathcal{L}\left(\sum_{i=1}^{n} w_i x_i\right)$$

where $\mathcal{L}$ is the (non-linear) activation function

We want to minimize the error $F(w_1, \ldots, w_n)$ but,

due to nonlinearity of $\mathcal{L}$, **now F is not convex**!



Minimizing the prediction error: Training vs Validation

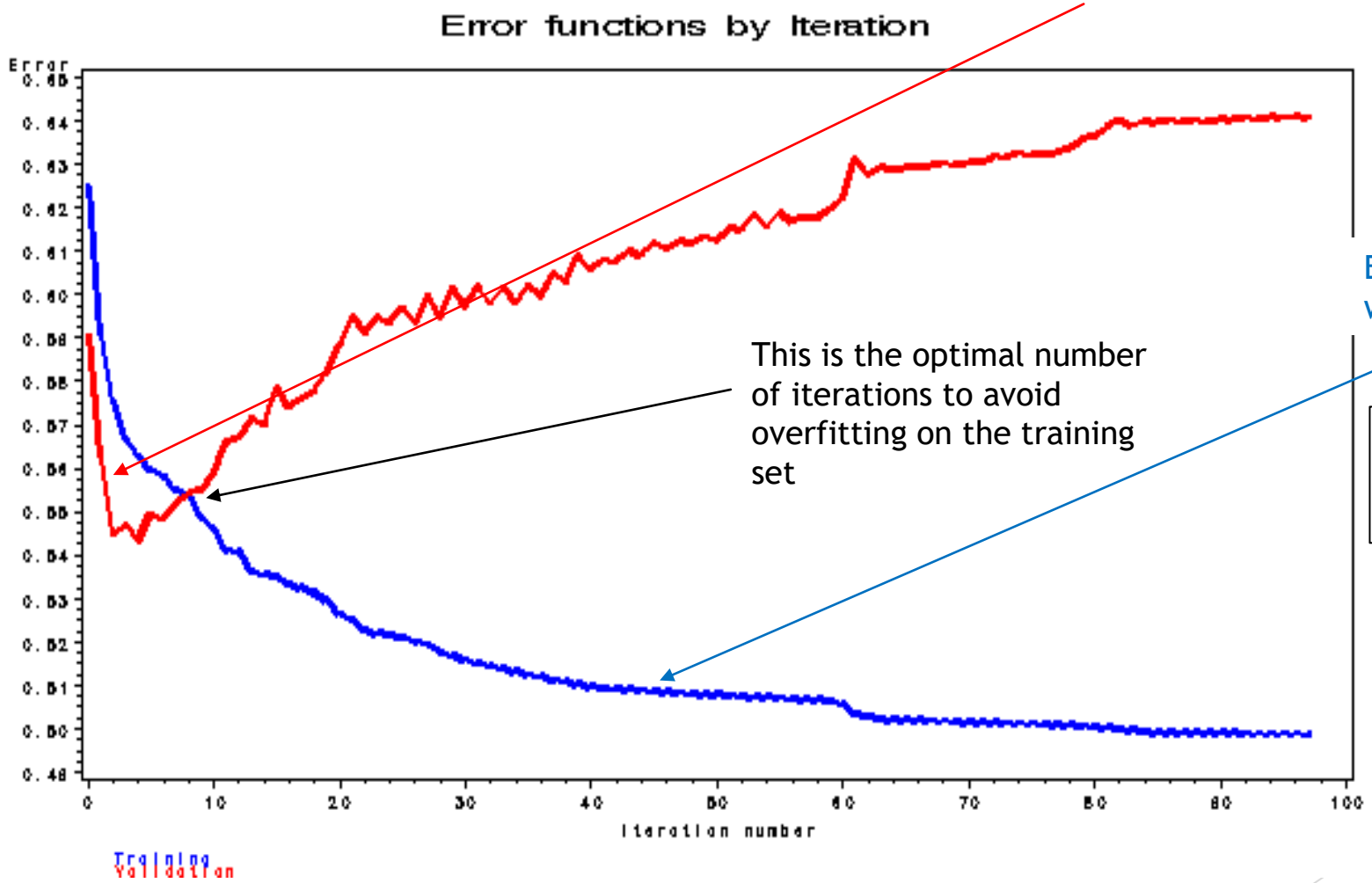
- The available dataset is usually divided into two sets: **Training set** and **Validation set**
- Training set is used to tune the weights w_i, with $i = 1,..,n$ by minimizing the prediction error
- Validation set is used to evaluate the ***generalization*** capability of the "trained" ANN
- Weigths are kept fixed during the Validation!
- Training a ANN – just like a Perceptron – is an iterative process, thus it is important to monitor both the two errors

General Learning Algorithm: Backpropagation

- 1: Randomly initialize all weights and thresholds (aka biases) and choose a "learning rate" η
- 2: Repeat until termination criteria satisfied
  - Give a training instance as input and "propagate" towards the output
  - Compute prediction error
  - Estimate the partial derivatives of the prediction error with respect to every weight
  - Update the weights with according to their derivatives and multiplied by η
- Basically, the error is "backpropagated" into the network and weights are updated depending on (1) how much they contribute to the prediction error and (2) the learning rate

An example…

Error on validation set initially decreases with the number of iterations, then increases…



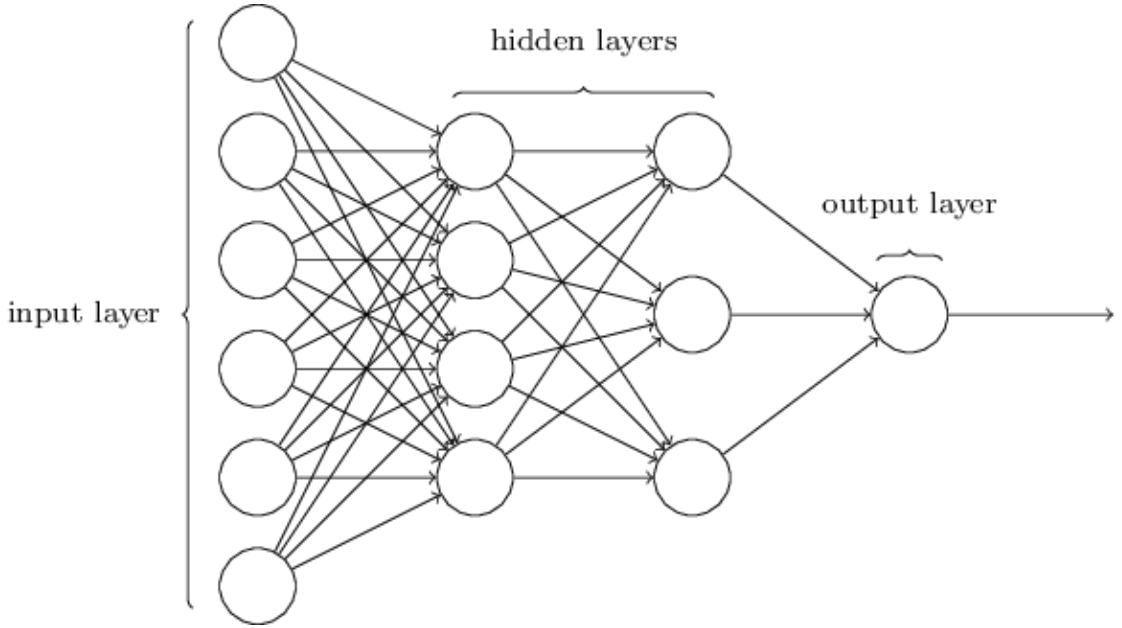
This is the optimal number of iterations to avoid overfitting on the training set

Error on training set decreases with the number of iterations

IMPORTANT!
Only training data are used to adjust weights and bias

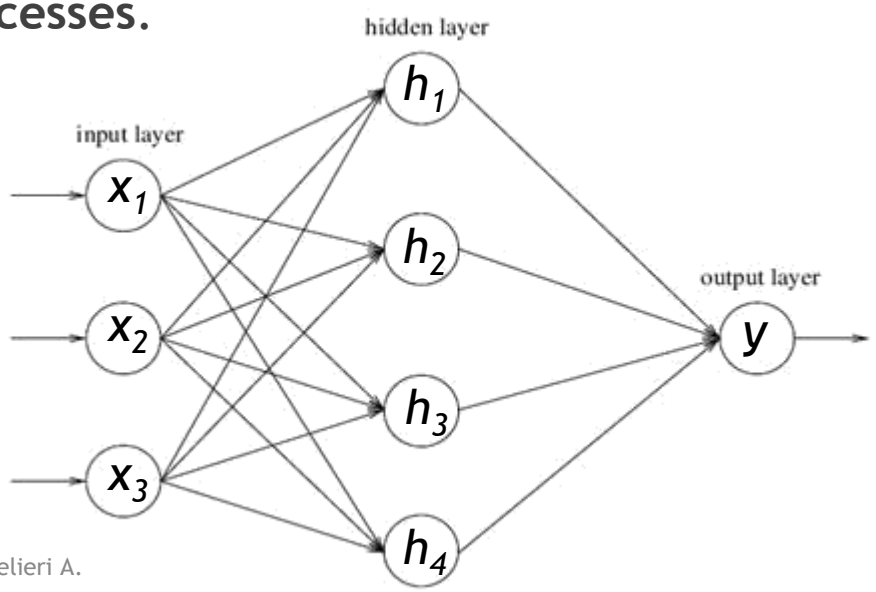
From Perceptron to Multi-layer Networks

- A single Perceptron is not sufficient to learn complex task
- As in the human brain, the power of ANN learning lays in connecting simple units between them
- Given the number of units, it is impracticable to consider *all the possible architectures*
- Simplification: units has to be organized into *consecutive layers* (multi-layer feed-forward networks)



Neural Networks

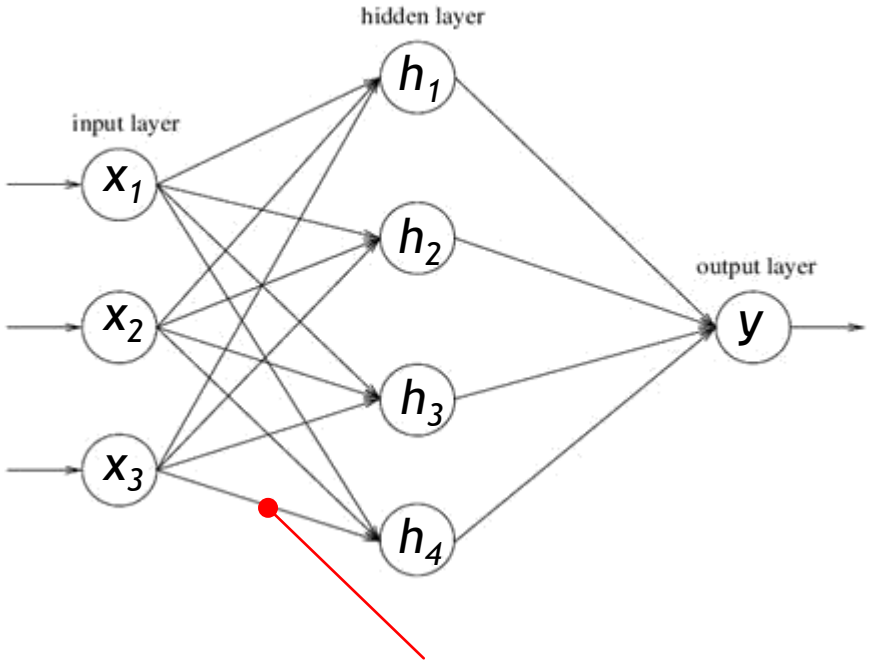
- The **feed-forward neural network** is one of the most popular methods to approximate a multivariate nonlinear function.
- Its distinctive feature is the **"hidden layers"**, in which input variables activate some nonlinear transformation functions producing a continuous signal to output nodes.
- This hidden layer scheme represents a very **efficient parallel distributed model of nonlinear statistical processes.**



Neural Networks

Each neuron h_i receives the input information and goes active, transmitting a signal to output, if a certain linear combination of the inputs, $\sum_{i \to j} \omega_{ij} x_i$, is greater than some threshold value α_j; that is, h_j transmits a non-zero value if

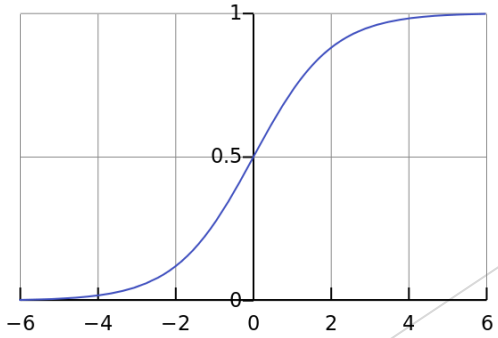
$$z_j = \sum_{i \to j} \omega_{ij} x_i - \alpha_j > 0$$



for instance this connection is weighted by w_{34}

where the sum is taken over all input nodes x_i feeding to neuron h_j, and ω_{ij} are the weights. The signal is produced by an activation function applied to z_j, which is usually taken as a logistic function,

$$\mathscr{L}(z) = \frac{\exp(z)}{1 + \exp(z)} = \frac{1}{1 + \exp(-z)}$$



Neural Networks

- Composing **a linear combination** of the possible values of **signals transmitted by the hidden layer**, plus **some value *a* to account for the bias** in the connection, with **an activation function *f* for the output node**, we obtain a mathematical expression for the basic feed-forward network with connections from inputs to hidden nodes, and from these to output:

$$y = f\left(a + \sum_{j=1}^{q} \theta_j \mathscr{L}\left(\alpha_j + \sum_{i \to j} \omega_{ij} x_i\right)\right)$$

Neural Networks and ARMA model

- For a more complex feed-forward network one allows **direct connections from input to output nodes**, and thus we have amore general expression for these neural networks

$$y = f\left(a + \sum_{i=1}^{p} \phi_i x_i + \sum_{j=1}^{q} \theta_j \mathscr{L}\left(\alpha_j + \sum_{i \to j} \omega_{ij} x_i\right)\right)$$

- If ***f*** is linear, we obtain an **ARMA model** for a given time series $\{y_t\}$. Indeed, taking $x_1 = y_{t-1}$, …, $x_p = y_{t-p}$ (i.e., as ***p*** different lags of the series), and the output $y = y_t$ as the time series value to forecast, and ***f*** linear, we have

$$y_t = a + \sum_{i=1}^{p} \phi_i y_{t-i} + \sum_{j=1}^{q} \theta_j \mathscr{L}\left(\alpha_j + \sum_{i \to j} \omega_{ij} y_{t-i}\right)$$

- The **moving average** part is being modelled **by a nonlinear function** on the input lags; this shows that the feed forward neural network is a generalization of ARMA(*p*, *q*) model.

A consideration…

- The case of ***f*** being a **threshold function** is also of interest in time series applications:
- If $y = r_t$ is a return to be computed, and $x_1, \dots, x_p$ are different lags of this return series $\{r_t\}$, or some other statistics (e.g., its historical volatility), then the equation with ***f*** threshold can be interpreted as **a model to forecast the direction of returns**, where $f(z) = 1$ implies positive returns and $f(z) = 0$ negative returns

This is Classification, not Regression any longer!

Classification performance measures

- **Accuracy:** the percentage of correctly classified examples w.r.t. the total number of examples
- **Recall (per class)**: as Accuracy but computed per class
- **Precision (per class)**: number of examples belonging to a class w.r.t. to the total number of examples predicted for that class
- Other… F1, ROC, AUC, etc.

		Reale	
		+	-
Predetto	+	40	0
	-	10	50

Accuracy = 40+50/100 = 90%
Recall "+" = 40/50 = 80%
Precision "+" = 40/40 = 100%

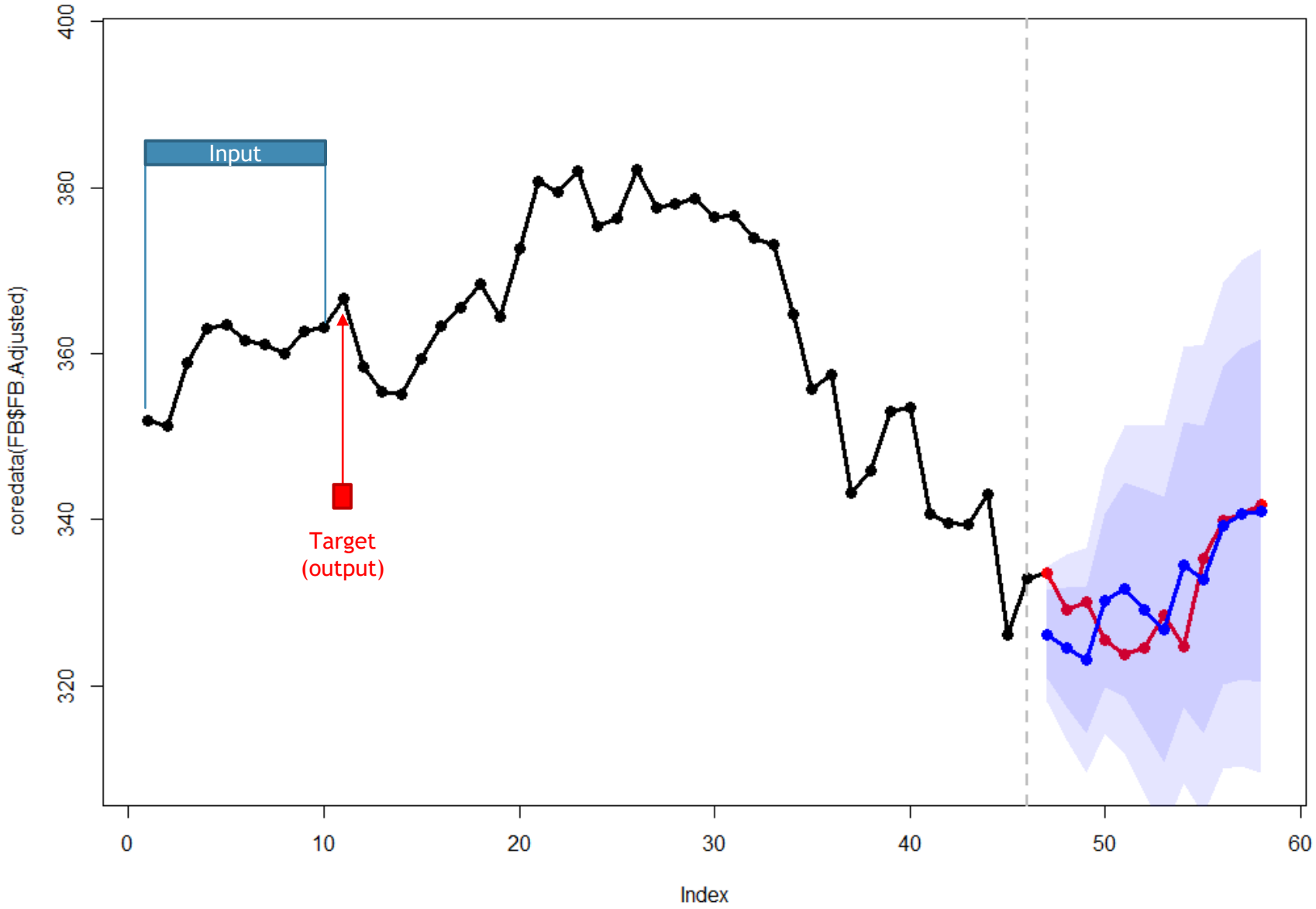
Recall "-" = 50/50 = 100%
Precision "-" = 50/60 = 83%

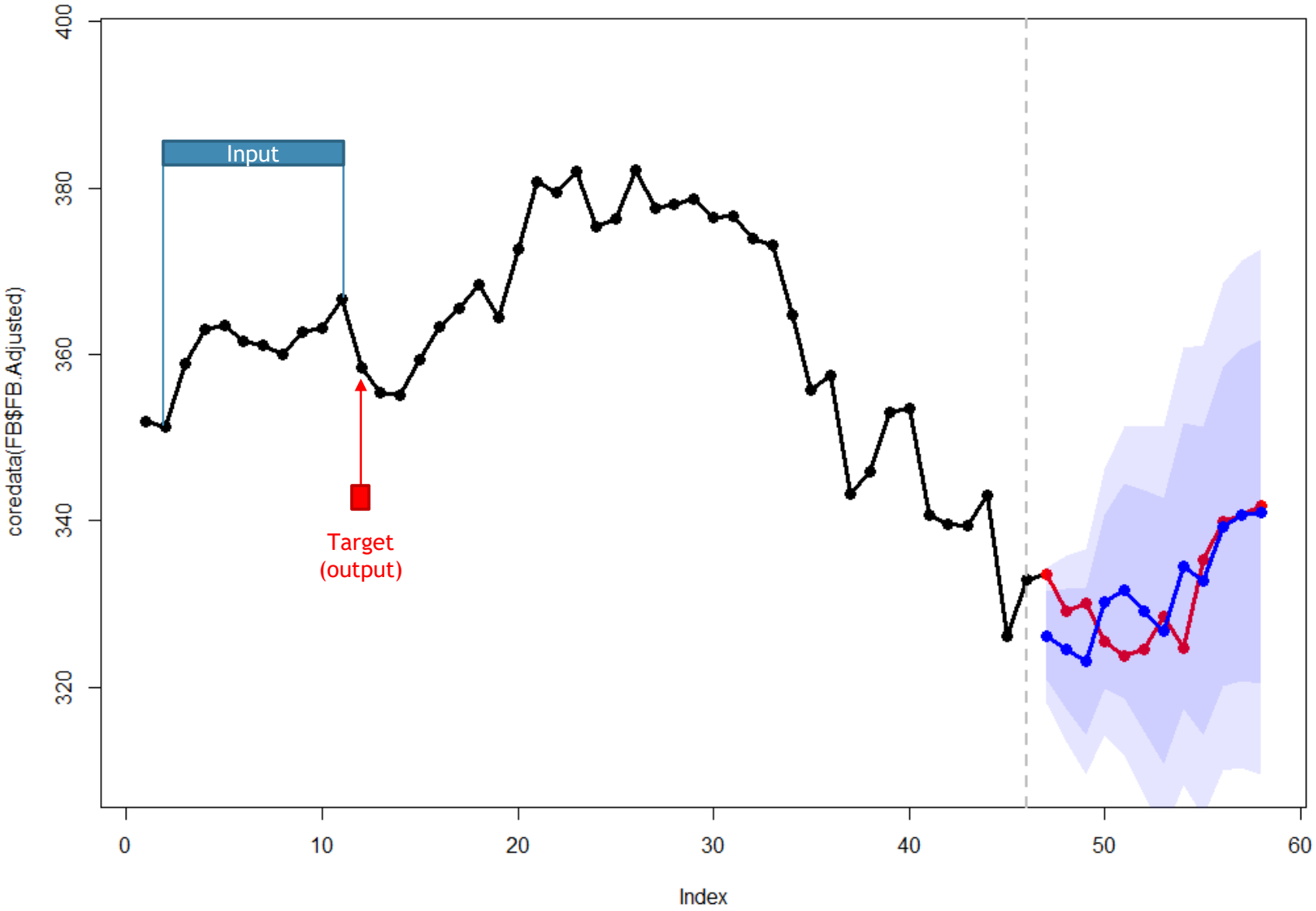
Learning the NNet regression model

- Learning a NNet model requires – in the most general setting – to split the available set of data in three different set:
  - training
  - validation
  - test
  - common splits are **60%-30%-10%** or **70%-20%-10%** (but others are possible)
- The goal is to find the optimal weights and bias to minimize some error measure, usually the mean squared error:

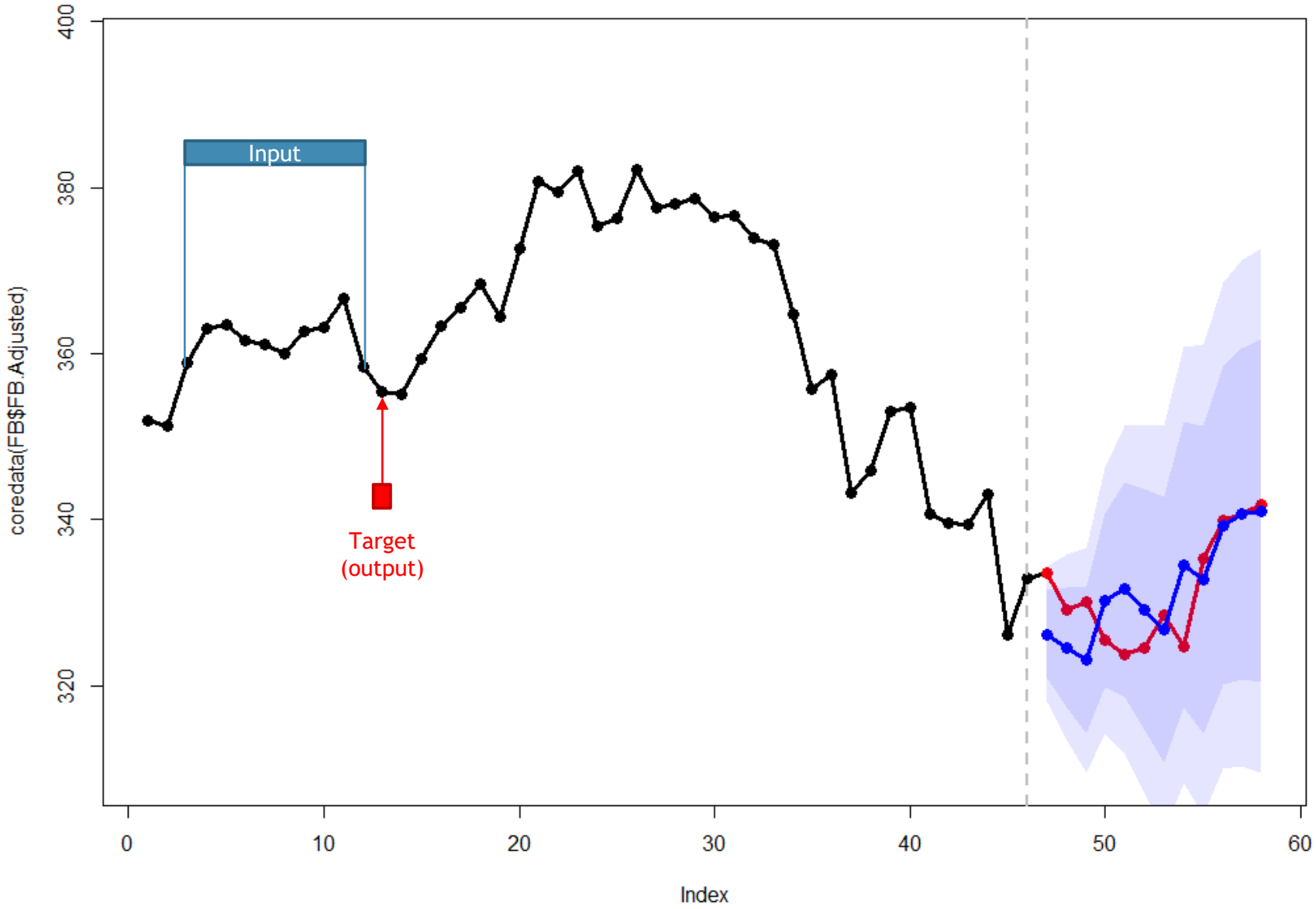
$$R(w) = \frac{1}{T}\sum_{t=1}^{T}(r_t - y_t)^2$$

- the training set is used to **estimate weights and bias**, validation set is used to **avoid overfitting**, test set is used as **further validation** (but it is not used to update weights and bias)

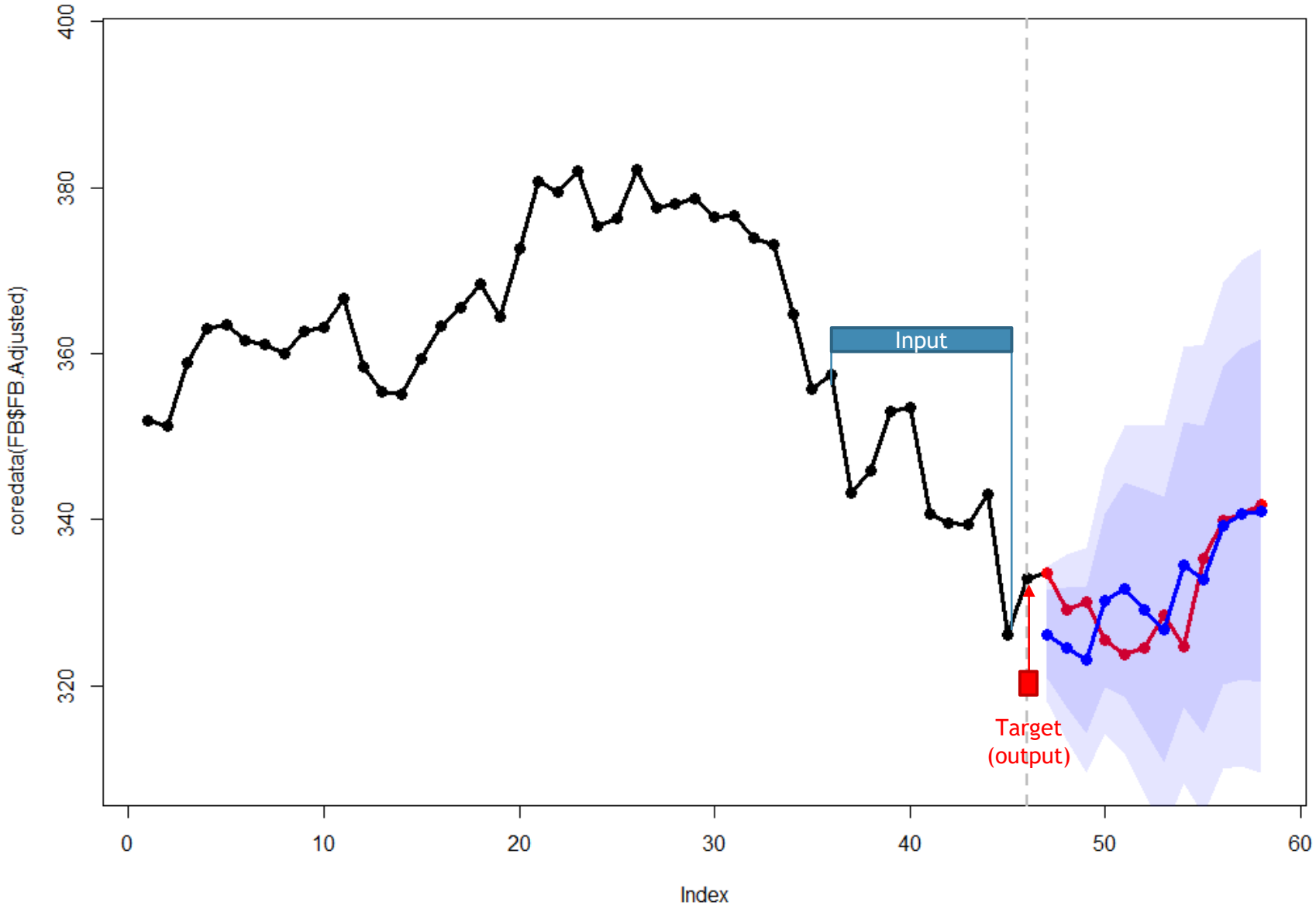

Input
Target
(output)
First data
coredata(FB$FB.Adjusted)
Index
400
380
360
340
320
0
10
20
30
40
50
60

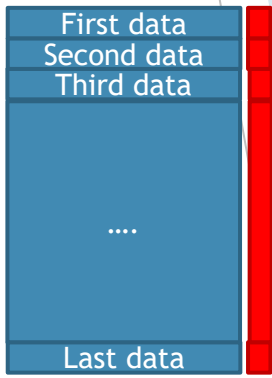


First data
Second data



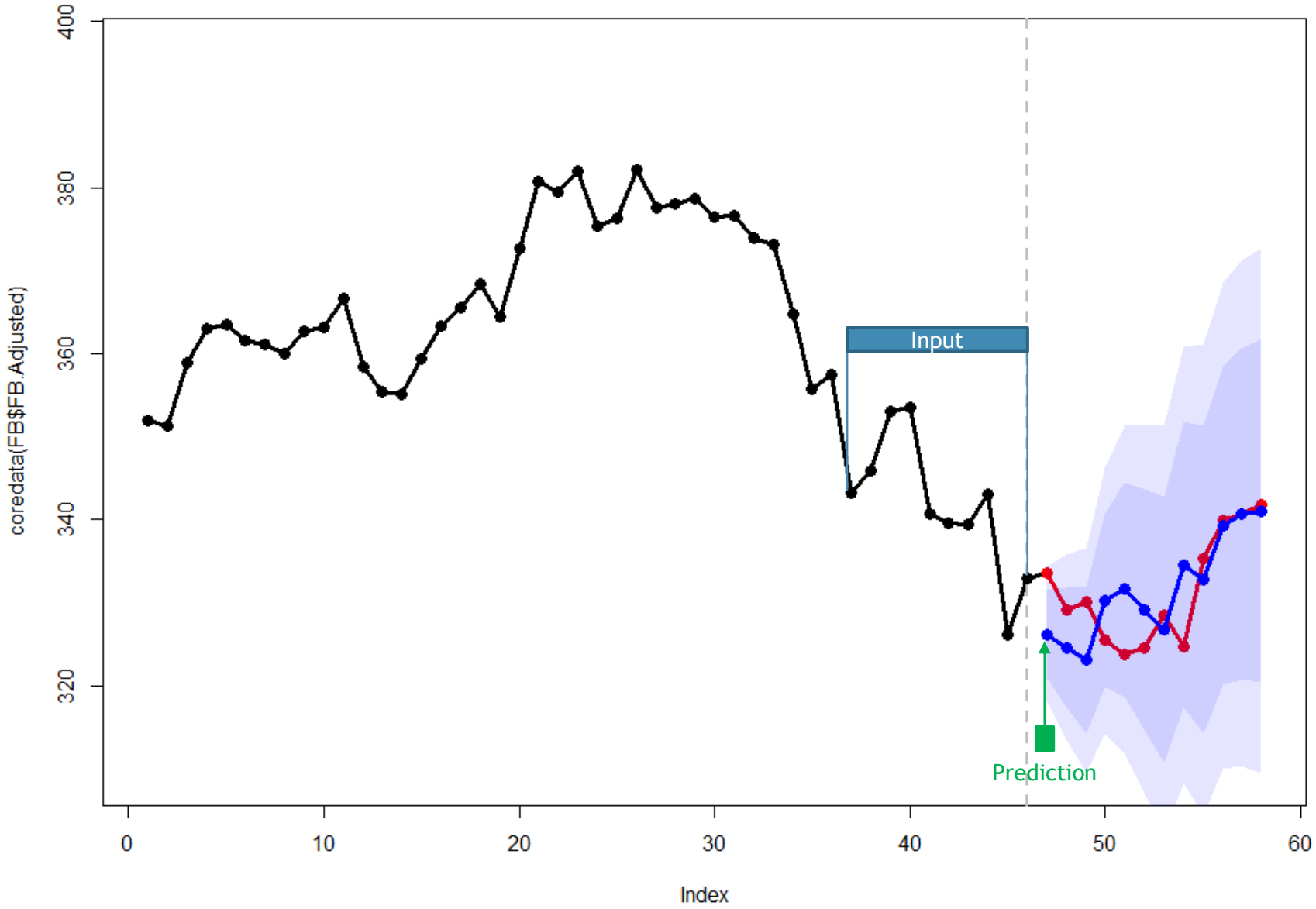
First data
Second data
Third data

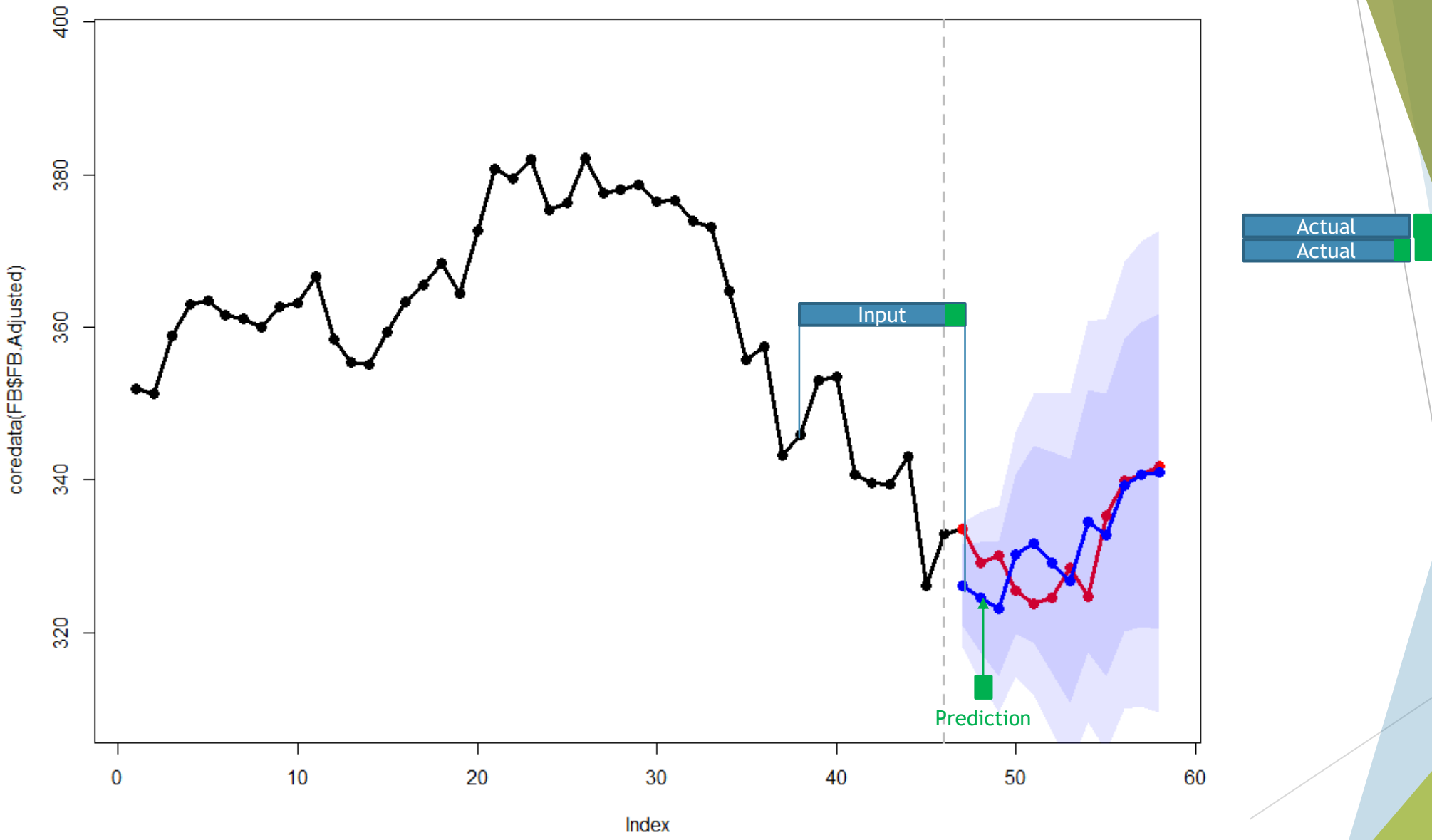


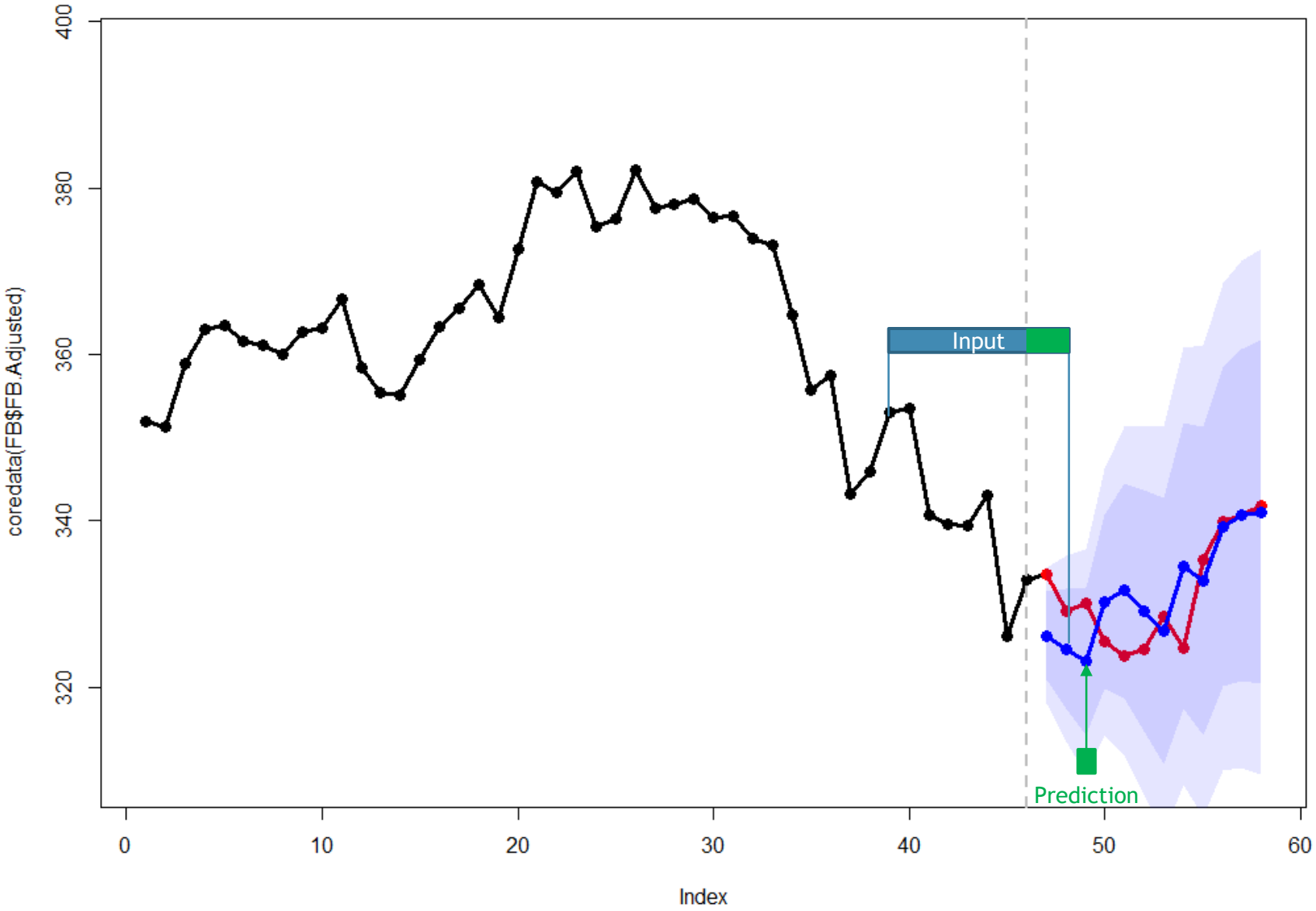


Training Set

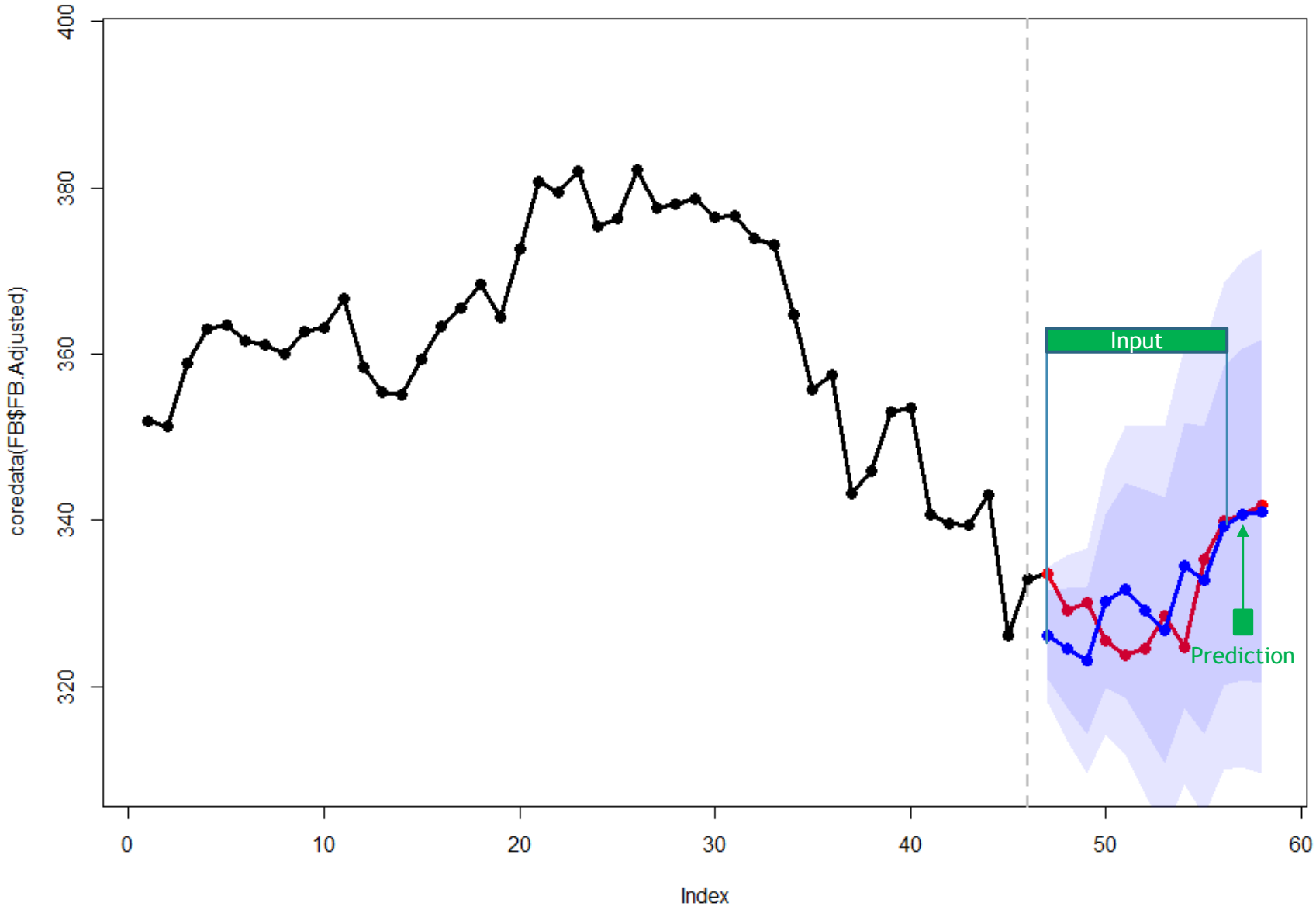
Resulting Model:
1-step ahead
forecasting!

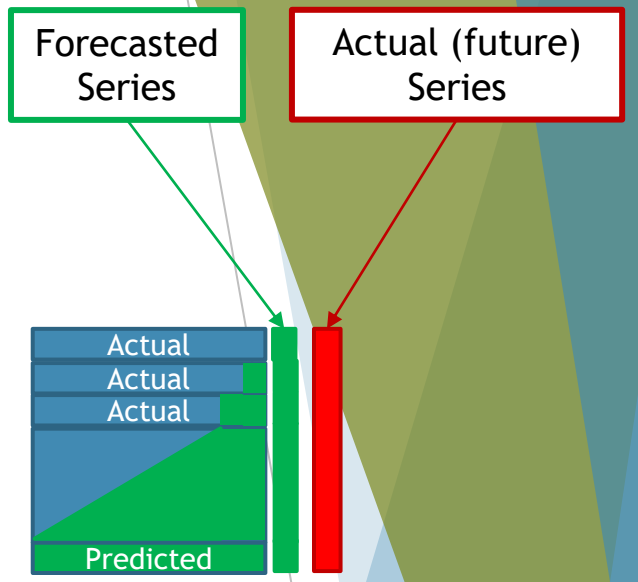

coredata(FB$FB.Adjusted)
400
380
360
340
320
0
10
20
30
40
50
60
Index
Input
Prediction
Actual


400
380
360
340
320
coredata(FB$FB.Adjusted)
0
10
20
30
40
50
60
Index
Input
Prediction
Actual
Actual


coredata(FB$FB.Adjusted)
400
380
360
340
320
0
10
20
30
40
50
60
Index
Input
Prediction


Actual
Actual
Actual





Recursive forecasting with 1-step ahead forecasting model!